Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing

Stanimire Tomov[1] and Jack Dongarra[1,2,3]

[1] University of Tennessee (USA)
[2] Oak Ridge National Laboratory (USA)
[3] University of Manchester (UK)

May 24, 2009

Abstract. We present a Hessenberg reduction (HR) algorithm for hybrid multicore + GPU systems that gets more than 16× performance improvement over the current LAPACK algorithm running just on current multicores (in double precision arithmetic). This enormous acceleration is due to proper matching of algorithmic requirements to architectural strengths of the hybrid components. The reduction itself is an important linear algebra problem, especially with its relevance to eigenvalue problems. The results described in this paper are significant because Hessenberg reduction has not yet been accelerated on multicore architectures, and it plays a significant role in solving nonsymmetric eigenvalue problems. The approach can be applied to the symmetric problem and in general, to two-sided matrix transformations. The work further motivates and highlights the strengths of hybrid computing: to harness the strengths of the components of a hybrid architecture to get significant computational acceleration which otherwise may have been impossible.
Keywords: Hessenberg reduction, eigenvalue problems, two-sided factorizations, dense linear algebra, hybrid computing, GPUs.

1 Introduction

Hardware trends. When processor clock speeds flatlined in 2004, after more than fifteen years of exponential increases, CPU designs moved to multicores. There is now widespread recognition that performance improvement on CPU-based systems in the near future will come from the use of multicore platforms. Along with multicores, the HPC community also started to use alternative hardware solutions that can overcome the shortcomings of standard multicores on a number of applications. One important example here is the use of Graphics Processing Units (or GPUs) for general purpose HPC. Graphics hardware has substantially evolved over the years, exponentially outpacing CPUs in performance, to the point where current GPUs have reached a theoretical peak performance of 1 TFlop/s in single precision, support fully the IEEE double precision arithmetic standard [20], and have a programming model (e.g. see CUDA [21]) that may revive the *quest for a free lunch* [16]. These developments have pushed the use

of GPUs to become pervasive [22, 32, 33]. Currently, major chip manufacturers, such as Intel, AMD, IBM and NVIDIA, make it more evident that future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of two major types of components:

1. Multi/many-cores, where the number of cores will continue to escalate; and
2. Special purpose hardware and accelerators, especially GPUs.

These trends motivate our work because in order to efficiently use the emerging hybrid hardware, optimal software solutions will themselves have to hybridize, or in other words, to match algorithmic requirements to architectural strengths of the hybrid components. Indeed, in this paper we show that although there are algorithmic bottlenecks that prevent the Hessenberg reduction from efficiently using a multicore architecture, a hybrid solution that relies on proper task splitting and task scheduling over the multicore and GPU components can overcome these bottlenecks and as a result to yield an enormous performance acceleration.

Hessenberg reduction. The reduction to upper Hessenberg form [15] is an important linear algebra problem, especially with its relevance to eigenvalue solvers. In particular, it is the first step in computing the Schur decomposition of a non-symmetric square matrix, which in turn gives, for example, the solution for the non-symmetric eigenvalue problem. The operation count for the reduction of an $n \times n$ matrix is approximately $\frac{10}{3}n^3$ which, in addition to not running efficiently on current architectures, makes the reduction a very desirable target for acceleration.

The bottleneck. The problem in accelerating HR stems from the fact that the algorithm is rich in Level 2 BLAS operations, which do not scale on multicore architectures and run only at a fraction of the machine's peak performance. There are dense linear algebra (DLA) techniques that can replace Level 2 BLAS operations with Level 3 BLAS. For example, in the so-called one-sided factorizations like LU, QR, and Cholesky, the application of consecutive Level 2 BLAS operations that occur in the algorithms can be delayed and accumulated so that at a later moment the accumulated transformation is applied at once as a Level 3 BLAS (see LAPACK [1]). This approach totally removes Level 2 BLAS from Cholesky, and reduces its amount to $O(n^2)$ in LU, and QR, thus making it asymptotically insignificant compared to the total $O(n^3)$ amount of operations for these factorizations. The same technique can be applied to HR [18], but in contrast to the one-sided factorizations, it still leaves about 20% of the total number of operations as Level 2 BLAS [26]. We note that 20% Level 2 BLAS may not seem much, but in practice, using a single core of a multicore machine, these 20% can take about 70% of the total execution time, thus leaving the grim perspective that multicore use – no matter how many cores would be available – can ideally reduce only the 30% of the execution time that are spent on Level 3 BLAS.

Current work directions. A subject of current research in the field of DLA is to design algorithms of minimized communications [4, 2], e.g. through designing algorithms where ideally all the operations involved are Level 3 BLAS. This is possible for the one-sided factorizations. Block Cholesky, as mentioned, already has this property, for QR certain out-of-core versions achieve this [17], and for LU there are similar to QR out-of-core algorithms [23], randomization techniques [3, 29], etc. These techniques are applied in current efforts for developing efficient algorithms for multicore architectures [10, 25]. Replacing Level 2 BLAS entirely with Level 3 BLAS is not possible for HR because the factorization is two-sided - namely, elementary Level 2 BLAS operations (that have to be accumulated as a Level 3 BLAS) have to be applied to both sides of the operator. The effect of this is that the accumulation of say nb Level 2 BLAS operations (where nb is a certain block size) as a Level 3 BLAS (in a process sometimes called *panel factorization*), still requires nb Level 2 BLAS operations. These operations involve the entire trailing matrix - e.g. for panel of size $i \times nb$ the Level 2 BLAS involve matrices of size $i \times i$ *vs* being restricted to just the $i \times nb$ panel for example as in the one sided factorizations.

The rest of the paper is organized as follows. In Section 2, we give background information on multicore and GPU-based computing in the area of DLA. Section 3 describes the standard HR algorithm and the proposed hybridization. Next are performance results (Section 4) and finally conclusions (Section 5).

2 Hybrid GPU-based computing

The development of high performance DLA for new architectures, and in particular multicores, has been successful in some cases, like the one sided factorizations, and difficult for others, like some two-sided factorizations. The situation is similar for GPUs - some algorithms map well, others do not. By combining these two architectures in a hybrid multicore + GPU system we seek to exploit the opportunity of developing high performance algorithms, as bottlenecks for one of the components (of this hybrid system) may not be for the other. Thus, proper work splitting and scheduling may lead to very efficient algorithms.

Previous work. This opportunity for acceleration has been noticed before in the context of one sided factorizations. In particular, while developing algorithms for GPUs, several groups [31, 5, 3] observed that panel factorizations are often faster on the CPU than on the GPU, which led to the development of highly efficient one sided hybrid factorizations for single CPU core + GPU [11, 30], multiple GPUs [30, 24], and multicore+GPU systems [29]. M. Fatica [13] developed hybrid DGEMM and DTRSM for GPU-enhanced clusters, and used them to accelerate the Linpack benchmark. This approach, mostly based on BLAS level parallelism, results only in minor or no modifications to the original source code.

Further developments. We found that a key to generalize and further develop the hybrid GPU-based computing approach is the concept of representing

algorithms and their execution flows as Directed Acyclic Graphs (DAGs). In this approach we split the computation in tasks and dependencies among them, and represent this information as a DAG, where DAG nodes are the tasks and DAG edges the dependencies (this was originally introduced in the context of DLA for just multicore architectures [9]). Figure 1 shows an illustration. The nodes in red in this case represent the sequential parts of an algorithm (e.g. panel factorization) and the ones in green the tasks that can be done in parallel (e.g. the update of the trailing matrix). Proper scheduling can ensure very efficient execution. This is the case for the one sided factorizations, where we schedule the execution of the tasks from the critical path on the CPU (that are in general small, do not have enough parallelism, and therefore could not have been efficiently executed on the GPU) and the rest on the GPU (grouped in large task for single kernel invocation as shown; highly parallel).

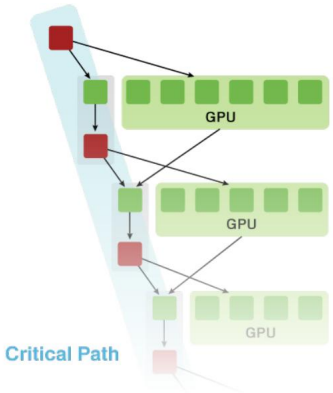

Fig. 1. Algorithms as DAGs for hybrid GPU-based computing

The hybrid approaches mentioned so far have used GPUs for Level 3 BLAS parts of their computation. We note that the introduction of GPU memory hierarchies, e.g. in NVIDIA's CUDA-enabled GPUs [33], provided the opportunity for an incredible boost of Level 3 BLAS [30, 19], because memory could be reused rather than having performance relying exclusively on high bandwidth as in earlier GPUs. Indeed, one can see that early attempts to port DLA on GPUs have failed to demonstrate speedup compared to CPUs [12, 14]. Nevertheless, high bandwidth has always been characteristic for GPUs, and can be instrumental in overcoming bandwidth bottlenecks in a number of very important DLA algorithms, as shown in this paper. We design a hybrid HR algorithm that exploits the strength of multicore and GPU architectures, where related to GPUs, we use their high performance on both Level 3 and Level 2 BLAS.

3 Hessenberg reduction

The HR algorithm reduces a general $n \times n$ matrix A to upper Hessenberg form H by an orthogonal similarity transformation $Q'AQ = H$. The matrix Q is represented as a product of $n-1$ elementary reflectors

$$Q = H_1\ H_2\ \ldots\ H_{n-1}, \quad H_i = I - \tau_i\ v_i v_i',$$

where τ_i is scalar, and v_i is a vector. In the block HR algorithm a set of nb reflectors, where nb is referred to as block size, can be grouped together

$$H_1\ H_2\ \ldots\ H_{nb} \equiv I - V\ T\ V',$$

where $V = (v_1|\ldots|v_{nb})$, and T is $nb \times nb$ upper triangular matrix. This transformation, known as *compact WY transform* [7, 27], is the basis for the delayed update idea mentioned above, where instead of applying nb Level 2 BLAS transformations (that are inefficient on current architectures), one can apply the accumulated transformation as a Level 3 BLAS. The resulting algorithm is known as block HR.

3.1 Block Hessenberg reduction

Algorithm 1 gives (in pseudo-code) the block HR, as currently implemented in LAPACK 3.1 (function `DGEHRD`; incorporating the latest improvements [26]). Function DGEHD2 on line 6 uses unblocked code to reduce the rest of the matrix.

Algorithm 1 DGEHRD(n, A)

1: **for** $i = 1$ to $n - nb$ step nb **do**
2: DLAHR2$(i,\ A(1:n,\ i:n),\ V,\ T,\ Y)$
3: $A(1:n,\ i+nb:n) -= Y\ V(nb+1:\ ,\ :\ )^T$
4: $A(1:i,\ i:i+nb) -= Y(1:i,\ :\ )V(1:nb,\ :\ )^T$
5: $A(i+1:n,\ i+nb:n) = (I - V\ T\ V^T)\ A(i+1:n,\ i+nb:n)$
6: **end for**
7: DGEHD2(...)

Algorithm 2 gives the pseudo-code for DLAHR2. DLAHR2 performs the two sided reduction for the current panel and accumulates matrices V and T for the *compact WY transform* $(I - V\ T\ V^T)$, and matrix $Y \equiv A(1:n,\ i:n)\ V\ T$. We denote by $Y_j \equiv (y_1|\ldots|y_j)$ the first j columns of Y, by T_j the submatrix $T(1:j,\ 1:j)$, and by $V_j \equiv (v_1|\ldots|v_j)$ the first j columns of V. Householder(j, x) returns a vector v and a scalar $\tau = v^T v/2$ where

$$v(1:j) = 0, \quad v(j+1) = 1, \quad v(j+2:\ ) = x(2:\ )/(x(1) + sign(x(1))||x||_2).$$

The latest improvement [26] is that in LAPACK 3.0 line 8 was distributed in the **for** loop above as a sequence of Level 2 BLAS operations.

Algorithm 2 DLAHR2(i, A, V, T, Y)

1: **for** $j = 1$ to nb **do**
2: $A(i+1:n,\ j)\ -=\ Y_{j-1}\ A(i+j-1,\ 1:j-1)$
3: $A(i+1:n,\ j)\ =\ (I - V_{j-1}\ T_{j-1}^T\ V_{j-1}^T)\ \ A(i+1:n,\ j)$
4: $[v_j,\ \tau_j]\ =\ \text{Householder}(j,\ A(i+j+1:n,\ j)\ )$
5: $y_j = A(i+1:n,\ j+1:n)\ v_j$
6: $T_j(1:j-1,\ j)\ =\ -\tau_j\ T_{j-1}\ V_{j-1}^T\ v_j; \quad T_j(j,j) = \tau_j$
7: **end for**
8: $Y(1:i,\ 1:nb) = A(1:i,\ i:n)\ V\ T$

3.2 On designing the hybrid algorithm

The following steps helped in designing the hybrid HR algorithm:

1. Locate the performance bottlenecks of the block HR;
2. Study the execution and data flow of the block HR;
3. Split the algorithm into a collection of tasks and dependencies among them;
4. Schedule the task execution over the components of the hybrid system;
5. Implement the design on high level: hybridizing LAPACK code with BLAS-based tasks on the multicore and CUBLAS-based tasks on the GPU.

Performance bottlenecks. For large applications, related to item 1 from the list, we recommend using tools like TAU [28] to help in locating performance bottlenecks. TAU can generate execution profiles and traces that can be analyzed with ParaProf [6], Jumpshot [35], and Kojak [34]. Profiles (of execution time or PAPI [8] counters) on runs using various numbers of cores of a multicore processor can be compared using ParaProf to easily see which functions scale, what is the performance for various parts of the code, what percent of the execution time is spent in the various functions, etc. In our case, using a dual socket quad-core Intel Xeon at 2.33 GHz, this analysis shows that line 5 of Algorithm 2 (colored in red) runs at about 70% of the total time, does not scale with multicore use, and has only 20% of the total amount of flops. We note that the computation at line 5 is the accumulation of the matrix Y and is obviously the major bottleneck of the block HR algorithm. These findings are also illustrated on Figure 2.

Task splitting. Studying the execution and data flow (item 2) is important in order to properly split the computation into tasks and dependencies (item 3). It is important here to study the memory footprint of the routines – what data is accessed and what data is modified – in order to accomplish item 3, and in particular to identify the algorithms' critical path, and decouple from it as much work as possible (tasks off of the critical path that in general would be trivial to parallelize). We identified that it is good to split every iteration of Algorithm 1 into three main tasks. Namely, these are tasks that we denote by P_i, M_i, and G_i, and associate them with the updates of 3 corresponding matrices. We denote

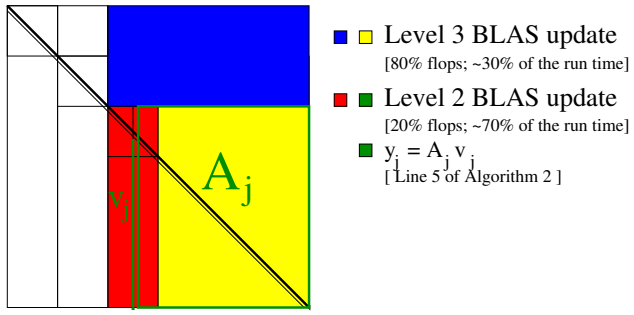

Fig. 2. Current computational bottleneck: the Level 2 BLAS $y_j = A_j v_j$

for convenience each of these matrices by the name of the task updating it, i.e. correspondingly P_i, M_i, and G_i. The splitting is illustrated on Figure 3, Left and described as follows:

- The panel factorization task P_i (20% of the flops; line 2 of Algorithm 1) updating the submatrix in red, and accumulating the matrices V_i, T_i and Y_i;
- Task G_i (60% of the flops) updating the submatrix in yellow

$$G_i = (I - V_i\ T_i\ V_i^T)\ G_i\ (I - V_i\ T_i\ V_i(nb+1:\ ,\ :\ )^T)$$

- Task M_i (20% of the flops) updating the submatrix in blue

$$M_i = M_i\ (I - V_i\ T_i\ V_i^T).$$

.

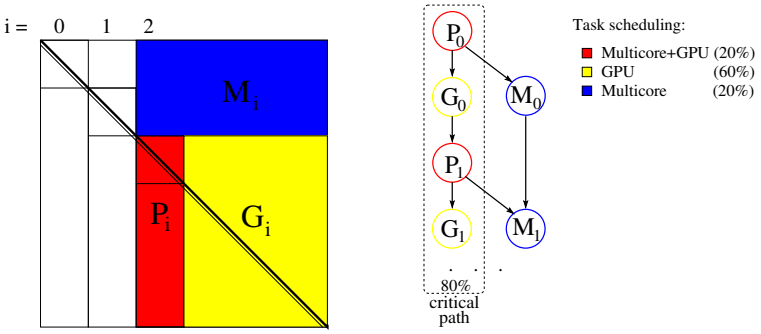

Fig. 3. Main tasks and their scheduling

We note that splitting line 3 of Algorithm 1 and merging it into tasks G_i and M_i is motivated by the memory footprint analysis. Indeed, using this splitting one

can see that task M_i gets independent of G_i and falls off the critical path of the algorithm (see Figure 3, Right). This is important for the next step: scheduling the tasks' execution over the components of the hybrid system. Note that the critical path is still 80% of the total amount of flops.

Scheduling. The scheduling is given also on Figure 3, Right. The tasks on the critical path must be done as fast as possible – and are scheduled in a hybrid fashion on both the Multicore and GPU. The memory footprint of task P_i, with 'P' standing for panel, is both P_i and G_i but G_i is accessed only for the time consuming computation of $y_j = A_j v_j$ (see Figure 2). Therefore, the part of P_i that is constrained to the panel (not rich in parallelism, with flow control statements) is scheduled on the multicore, and the time consuming $y_j = A_j v_j$ (highly parallel but requiring high bandwidth) is scheduled on the GPU. G_i, with 'G' standing for GPU, is scheduled on the GPU. This is Level 3 BLAS update and can be done very efficiently on the GPU. Moreover, note that G_{i-1} contains the matrix A_j needed for task P_i, so for the computation of $A_j v_j$ we have to only send v_j to the GPU and the resulting y_j back from the GPU to the multicore. The scheduling so far heavily uses the GPU, so in order to make the critical path execution faster and at the same time to make a better use of the multicore, task M_i, with 'M' standing for multicore, is scheduled on the multicore.

Hybrid code development. Finally, item 5 is a general recommendation that we advocate. This general approach makes the transition from LAPACK algorithms easier, algorithms more readable, and abstracts the resulting hybrid algorithms from the specifics of the GPU and its link to the multicore. Note that one does not need to know about GPUs or even CUDA in order to develop these algorithms, but can still design hybrid algorithms that use the available GPUs and get very high performance. The approach assumes that basic kernels (line CUBLAS) would be available from a third party, which is a mild assumption based on the current high pace of GPU kernels development in the scientific community.

3.3 Hybrid Hessenberg reduction

Algorithm 3 gives in pseudo-code the hybrid HR algorithm. Prefix 'd', standing for device, before a matrix denotes that the matrix resides on the GPU memory. The algorithm name is prefixed by MAGMA, standing for *Matrix Algebra for GPU and Multicore Architectures*, and denoting a project[4] on the development of a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current Multicore+GPU systems [2].

Algorithm 4 gives the pseudo-code for MAGMA_DLAHR2.

Figure 4 illustrates the communications between the multicore and GPU for inner/outer iteration j/i.

[4] see http://icl.cs.utk.edu/magma/

Algorithm 3 MAGMA_DGEHRD(n, A)

1: Send matrix A from the CPU to matrix dA on the GPU
2: **for** $i = 1$ to $n - nb$ step nb **do**
3: MAGMA_DLAHR2(i, V, T, dP_i, dV, dT, dY)
4: Send d$G_{i-1}(1 : nb, \ :)$ to the multicore (asynchronously)
5: Schedule G_i on the GPU (asynchronously; using dV, dT, and dY)
6: Schedule M_i on the multicore (asynchronously; using V and T)
7: **end for**
8: MAGMA_DGEHD2(...)

Algorithm 4 MAGMA_DLAHR2(i, V, T, dP_i, dV, dT, dY)

1: Send dP_i from the GPU to P on the multicore
2: **for** $j = 1$ to nb **do**
3: $P(\ : \ , \ j) \ -= \ Y_{j-1} \ T_{j-1} \ P(j-1, \ 1 : j-1)$
4: $P(\ : \ , \ j) \ = \ (I - V_{j-1} \ T_{j-1}^T \ V_{j-1}^T) \ \ P(\ : \ , \ j)$
5: $[v_j, \ \tau_j] \ = \ $ Householder$(j, \ P(j+1 : \ , \ j) \ )$
6: Send v_j from the multicore to dv_j on the GPU
7: d$y_j = \ $ d$A(i+1 : n, \ j+1 : n) \ $ dv_j
8: $T_j(1 : j-1, \ j) \ = \ -\tau_j \ T_{j-1} v_j; \quad T_j(j, j) = \tau_j$
9: Send dy_j from the GPU back to y_j on the CPU
10: **end for**
11: Send T from the multicore to dT on the GPU

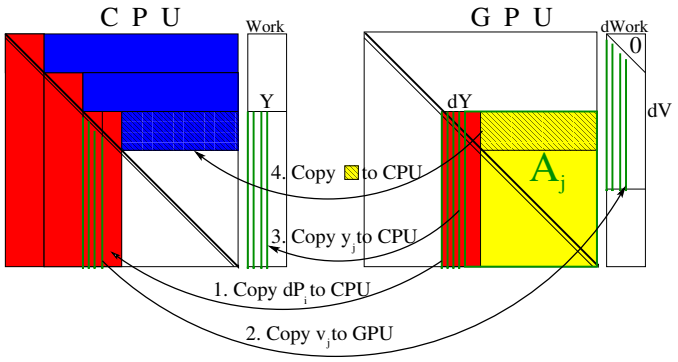

Fig. 4. CPU/GPU communications for inner/outer iteration j/i.

3.4 Differences with LAPACK

Our user interface is exactly as LAPACK's DGEHRD. The user gives and receives the factored matrix in the same format. The result is the same up to round-off errors related to a slightly different order of applying certain computations. In particular, LAPACK's matrix-matrix multiplications involving V are split into 2 multiplications: a DTRMM with the lower triangular sub-matrix $V(1 : nb, \ 1 : nb)$ and a DGEMM with the rest of V. As nb is usually small, multiplications on the GPU with triangular $nb \times nb$ matrices is slow. Therefore,

we keep zeroes in the upper triangular part of $V(1:nb,\ 1:nb)$ and perform multiplications with V using just one kernel call. For the same reason, multiplications with T are performed as DGEMMs. In LAPACK, matrix $Y = A\ V\ T$ is accumulated during the panel factorization. We accumulate $A\ V$ during the panel factorization and T is applied at once as a DGEMM during the matrix update part of the algorithm. Our work space is twice larger than LAPACK's work space on both the multicore and the GPU. This means we need work space of size $2 \times n \times nb$. On the multicore the additional space is used to enable processing tasks P and M in parallel (as each of them needs $n \times nb$ work space). On the GPU the additional space is used to separately store V from the matrix so that we can put zeroes only once in its upper triangular part, and use V as mentioned above. These modifications, in addition to providing higher performance, make also the code very readable, and shorter than LAPACK's.

4 Performance Results

The performance results that we provide in this section use NVIDIA's GeForce GTX 280 GPU and its multicore host, a dual socket quad-core Intel Xeon running at 2.33 GHz. On the multicore we use LAPACK and BLAS from MKL 10.0 and on the GPU CUBLAS 2.1, unless otherwise noted.

Performance. Figure 5 shows the performance of 2 hybrid HR algorithms, and the block HR on single core and multicore. The basic hybrid HR is for 1 core +

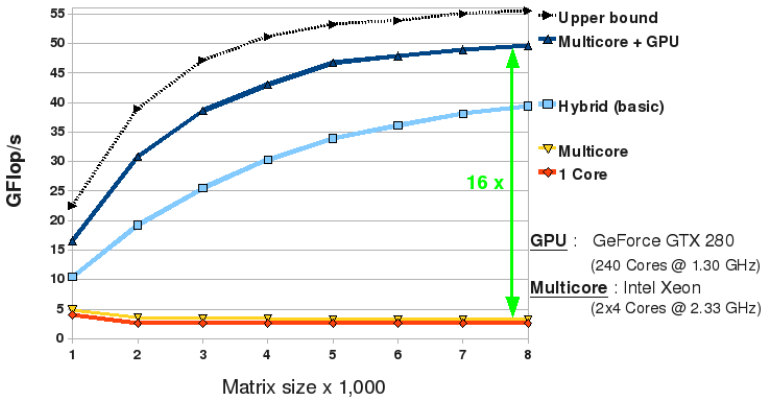

Fig. 5. Performance (in double precision) for the hybrid HR.

GPU, and uses CUBLAS 2.1. The Multicore+GPU hybrid algorithm is the one described in the paper plus various kernels' optimizations, described as follows. The result shows that we achieve an enormous 16× speedup compared to the current block HR running on multicore. We see that the basic implementation

brings most of the acceleration which is due to the use of hybrid architecture and the proper algorithmic design – splitting the computation into tasks, and their scheduling so that we match algorithmic requirements to architectural strengths of the hybrid components. We note that the performance achieved is half the performance of one-sided factorizations based (entirely) on Level 3 BLAS, e.g. the hybrid LU [29] runs on the same platform at up to 100 GFlop/s.

Performance upper bound We get asymptotically within 90% of the "upper bound" performance, as shown on Figure 5. Here upper bound denotes the performance of the *critical path* of our algorithm when we do not count synchronization and data transfer times, i.e. this is the performance of tasks P_i and G_i (without counting M_i) just based on the performance of the BLAS used.

Optimizations We optimized the GPU matrix-vector product as it is critical for the performance. Figure 6 compares cublasDgemv, MAGMA_DGEMV, and MKL DGEMV. We also give some implementation details at the bottom of the figure. The theoretically optimal implementation would have a perfor-

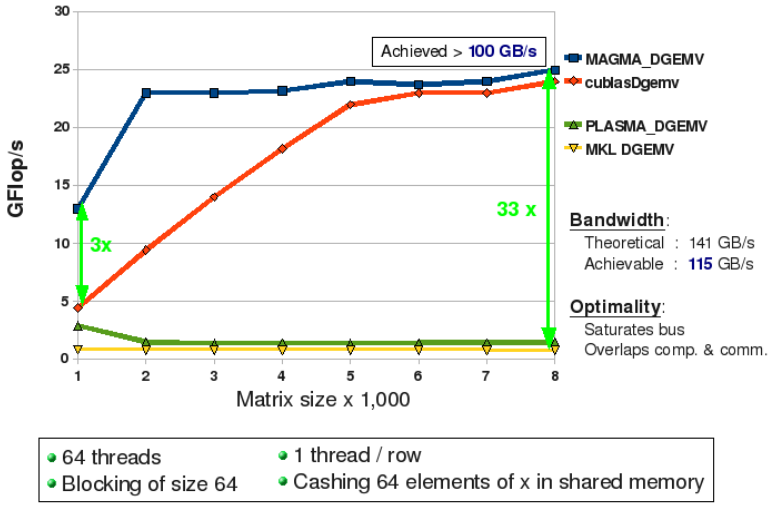

Fig. 6. Developing optimal GPU matrix-vector product.

mance of about 35 GFlop/s (the theoretical maximum bandwidth of 141 GB/s over 4). This would assume 100% bus utilization and 100% overlap of the computation with the communication needed. We achieve 25 GFlop/s. Shown is also the performance of a similar to the GPU algorithm but just for multicores (PLASMA_DGEMV) that uses 4 threads to achieve about 80% performance improvement over MKL's DGEMV. We can get close to optimal performance with a single thread for matrices stored in row-major format.

Using the optimized DGEMV gave up to 42% (achieved for matrices around 2000×2000). Using multicores *vs* single core gave a speedup of up to 16% (achieved for matrices in the range of $n = 2000$ to 3000).

We use block size $nb = 32$. Testing with larger nb gives slower performance results. For $nb = 32$ we used MAGMA_DGEMM kernels that outperform CUBLAS 2.1 by 10 GFlop/s on average. These kernels are based on the auto-tuning approach described in [19].

We also optimized the multicore implementation of tasks M_i. Our original implementation used MKL's parallel BLAS to get an average performance of about 17 GFlop/s for matrix of size $8,000$ (the averages for P_i and G_i are correspondingly 23 GFlop/s and 64 GFlop/s), and about 10 GFlop/s towards the end of the computation. We changed this to a 1-D block row partitioning of M_i and assigned the update for single block of rows to a single core. This is a trivial splitting and was easy to code using OpenMP. The performance improved to an average of 30 GFlop/s and up to 45 GFlop/s towards the end of the computation. High performance towards the end of the computation is important (especially for large matrices) because this is when M_i becomes larger and larger compared to P_i and G_i. Using the optimized code on a matrix of size $8,000$, the execution of tasks M_i is totally overlapped with the execution of P_i and G_i for the first 97% of the computation, and becomes dominant in the last 3% of the computation. In our case this was not a bottleneck because of the high performance that we achieve at the end. Another solution is if the GPU is scheduled to do part of M_i near the end of the computation.

5 Conclusions

We presented a hybrid HR algorithm that gets $16\times$ performance improvement over the latest LAPACK 3.1 algorithm running just on current multicores (in double precision arithmetic). This result is significant because the Hessenberg reduction has not been accelerated on multicore architectures, and it plays a significant role in solving nonsymmetric eigenvalue problems. Moreover, our approach shows a way of accelerating a large and important class of DLA algorithms, namely the two-sided factorizations.

The approach taken also makes a point that is of general interest to hybrid computing: the complexity of development could be kept low if one works on a high level, but at the same time it is still possible to get significant acceleration by properly splitting the computation into tasks (as implemented in already existing high performance libraries) and properly scheduling them in order to match algorithmic requirements to architectural strengths of the hybrid components.

Acknowledgments

This work is supported by Microsoft, the U.S. National Science Foundation, and the U.S. Department of Energy. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations. We thank Julien Langou

(UC, Denver) and Hatem Ltaief (UT, Knoxville) for their valuable suggestions and discussions on the topic.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.
2. M. Baboulin, J. Demmel, J. Dongarra, S. Tomov, and V. Volkov, *Enhancing the performance of dense linear algebra solvers on GPUs [in the MAGMA project]*, Poster at supercomputing 08, November 18, 2009.
3. M. Baboulin, J. Dongarra, and S. Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures*, Tech. report, LAPACK Working Note 200, May 2008.
4. G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in linear algebra*, Tech. report, LAPACK Working Note 218, May 2009.
5. S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Ortí, *Solving dense linear systems on graphics processors*, Technical Report ICC 02-02-2008, Universidad Jaime I, February, 2008.
6. R. Bell, A. Malony, and S. Shende, *ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis*, Euro-Par, 2003, pp. 17–26.
7. C. Bischof and C. Van Loan, *The WY representation for products of Householder matrices*, SIAM J. Sci. Stat. Comp. **8** (1987), no. 1, S2–S13, Parallel processing for scientific computing (Norfolk, Va., 1985). MR 88f:65070
8. S. Browne, C. Deane, G. Ho, and P. Mucci, *PAPI: A portable interface to hardware performance counters*, (June 1999).
9. A. Buttari, J. Dongarra, J. Kurzak, J. Langou, and S. Tomov, *The impact of multicore on math software*, In PARA 2006, Umea Sweden, 2006.
10. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Technical Report UT-CS-07-600, University of Tennessee, 2007, LAPACK Working Note 191.
11. J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie, *Exploring new architectures in accelerating CFD for Air Force applications*, Proc. of HPCMP UGC08 (July 14-17, 2008), http://www.cs.utk.edu/~tomov/ugc2008_final.pdf.
12. K. Fatahalian, J. Sugerman, and P. Hanrahan, *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*, HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA), ACM, 2004, pp. 133–137.
13. M. Fatica, *Accelerating Linpack with CUDA on heterogenous clusters*, GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), ACM, 2009, pp. 46–51.
14. N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, *LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware*, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2005, p. 3.
15. G. H. Golub and C. F. Van Loan, *Matrix computations*, second ed., Baltimore, MD, USA, 1989.

16. W. Gruener, *Larrabee, CUDA and the quest for the free lunch*, http://www.tgdaily.com/content/view/38750/113/, 08/2008, TGDaily.
17. B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Trans. Math. Softw. **31** (2005), no. 1, 60–78.
18. S. Hammarling, D. Sorensen, and J. Dongarra, *Block reduction of matrices to condensed forms for eigenvalue computations*, J. Comput. Appl. Math **27** (1987), 215–227.
19. Y. Li, J. Dongarra, and S. Tomov, *A note on auto-tuning GEMM for GPUs.*, Tech. report, LAPACK Working Note 212, January 2009.
20. NVIDIA, *Nvidia Tesla doubles the performance for CUDA developers*, Computer Graphics World (06/30/2008).
21. NVIDIA, *NVIDIA CUDA Programming Guide*, 6/07/2008, Version 2.0.
22. J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. Purcell, *A survey of general-purpose computation on graphics hardware*, Computer Graphics Forum **26** (2007), no. 1, 80–113.
23. E. Quintana-Ortí and R. van de Geijn, *Updating an LU factorization with pivoting*, ACM Trans. Math. Softw. **35** (2008), no. 2, 1–16.
24. G. Quintana-Ortí, F.Igual, E.Quintana-Ortí, and R. van de Geijn, *Solving dense linear systems on platforms with multiple hardware accelerators*, PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (New York, NY, USA), ACM, 2009, pp. 121–130.
25. G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. Van Zee, and R. van de Geijn, *Programming algorithms-by-blocks for matrix computations on multithreaded architectures*, Technical Report TR-08-04, University of Texas at Austin, 2008, FLAME Working Note 29.
26. G. Quintana-Ortí and R. van de Geijn, *Improving the performance of reduction to Hessenberg form*, ACM Trans. Math. Softw. **32** (2006), no. 2, 180–194.
27. R. Schreiber and C. Van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Stat. Comp. **10** (1989), no. 1, 53–57. MR 90b:65076
28. S. Shende and A. Malony, *The TAU parallel performance system*, Int. J. High Perform. Comput. Appl. **20** (2006), no. 2, 287–311.
29. S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems.*, Tech. report, LAPACK Working Note 210, October 2008.
30. V. Volkov and J. Demmel, *LU, QR and Cholesky factorizations using vector capabilities of GPUs*, Tech. Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.
31. V. Volkov and J. W. Demmel, *Using GPUs to accelerate linear algebra routines*, Poster at PAR lab winter retreat, January 9, 2008, http://www.eecs.berkeley.edu/ volkov/volkov08-parlab.pdf.
32. *General-purpose computation using graphics hardware*, http://www.gpgpu.org.
33. *NVIDIA CUDA ZONE*, http://www.nvidia.com/object/cuda_home.html.
34. F. Wolf and B. Mohr, *Kojak - a tool set for automatic performance analysis of parallel applications*, Proc. of Euro-Par (Klagenfurt, Austria), Lecture Notes in CS, vol. 2790, Springer, August 2003, pp. 1301–1304.
35. O. Zaki, E. Lusk, W. Gropp, and D. Swider, *Toward scalable performance visualization with Jumpshot*, HPC Applications **13** (1999), no. 2, 277–288.